RESEARCH ON ANDROID BROWSER FUZZING BASED ON BITMAP STRUCTURE

Yuanwei Hou, Guo Tao, Zhiwei Shi, Liu Juan

China Information Techonology Security Evaluation Center, Beijing 100085, China
houyw@itsec.gov.cn, guot@itsec.gov.cn, shizw@itsec.gov.cn,yuhua-7@163.com

Abstract: As one of the most important methods of vulnerability mining, the technology of fuzzing is developing rapidly and widely used during vulnerability analysis of various kinds of software. Nowadays, with the widespread craze of Android phones, the security of Android operating system has drawn more and more attention. However, fuzzing technology of Android is still in the beginning and professional fuzzing products of Android are unavailable. In this article, a fuzzing method of Android browser based on bitmap structure is presented. The result of experiment proves that this method is efficient in triggering Android browser crashes and can help with further work of analysis and exploitation.

Keywords: Android browser; Fuzzing; Bmp

1 Introduction

Developed by the Open Handset Alliance (visibly led by Google), Android is a widely anticipated open source operating system, which provides user-friendly GUI for the consumers and convenient developing and debugging framework for developers. Nowadays, Android OS has gained the most mobile market share all over the world. In United States, the market shared reached 51% [1] and in China, it's been over 76%, which means you can find one using Android OS every two mobile phones [2]. As people tend to keep private and sensitive information in their mobile phones, the security of mobile operating system has become their major concern, which makes it necessary to take a deep research on the Android operating system, to find and analyze the vulnerabilities in order to make it more secure and safe.

Since the birth of Android, security issues and system vulnerabilities has never been out of people's sight [3-5]. However, as the fuzzing theories of smart phones have not been mature and professional fuzzing tools or framework are not available, this article described a method of how to find vulnerabilities of Android Browser by fuzzing using the framework and tools provided by Android operation system. This fuzzing method is not blind fuzzing but based on the file structure of bitmap focusing on the modules of Android browser that parses bitmap files. This method generates effective mal-formed data based on the vulnerable points of Android browser, automatically inputs mal-formed data and keeps track of exceptions and crashes for later analysis and further exploitation.

978-1-4673-1857-0/12/$31.00 ©2012 IEEE

2 Related work

2.1 Fuzzing framework

It is common to use fuzzing technique to test target programs for vulnerability [6]. The methodology is useful against large applications, where any bug affecting memory safety is likely to be a severe vulnerability [7].

Normally a fuzzer consists of three parts: Mal-formed data generator, Mal-formed test cases reader and Exception monitor. As in Figure 1, firstly the generator would produce test cases containing the mal-formed data, and then the reader will automatically or manually read each test case into the target application. If the application runs as expected then next test case will be read until the application crashes or throws exceptions. Then the monitor would record the test case number and all necessary information when the crash happens for further analysis or exploitation. The fuzzing method discussed in this article is also under such framework, the fuzzer to be shown in this article is also composed of such three parts.

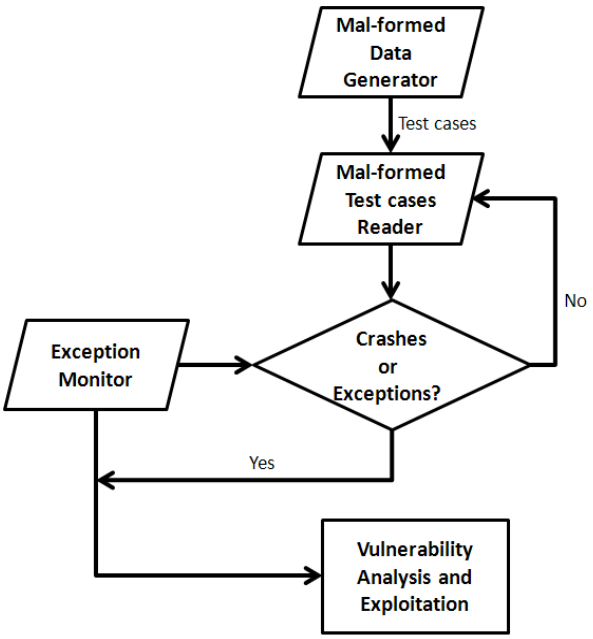

Figure 1 General fuzzing framework

2.2 Historical vulnerabilities

The browser built into Android operation system is based on WebKit. WebKit is an open source web browser engine [www.webkit.org], which began in 1998 as the KDE's HTML layout engine KHTML and KDE's JavaScript engine (KJS)[8]. As WebKit contains features as open-source, focusing on high performance browser, broad compatibility and transportability, it is widely adopted in mobile phones such as Android, iPhone and Nokia Symbian phones. The web framework engine of Android browser uses WebCOre and JSCore of WebKit project for web page arrangement. Nevertheless, WebKit framework is not so secure and robust. Researchers have found lots of vulnerabilities and security risks of WebKit. We can find more than 400 vulnerability issues about WebKit on CVE. The reasons that resulted in the vulnerabilities are various, but mainly happen when WebKit engine parses the HTML, CSS, JavaScript objects and DOM objects. As Android browser is based on WebKit engine, it is inevitable to be affected by the vulnerabilities. Therefore, it is really necessary to do deeper research about these vulnerabilities and get some conclusion.

Among the four aspects that vulnerabilities may be found about WebKit as mentioned above, this article chose HTML parsing problem to discuss, and the other three aspects can do exactly the same research as this one. There many kinds of tags in HTML, you can insert a picture in an HTML page using tag<img />:

```
<img src="1.bmp" width="104" height="142" />
```

This article has analyzed the vulnerabilities about BMP file parsing. There are 82 published vulnerability issues on CVE, covering various platforms and software. The vulnerable points which lead to these vulnerabilities are mainly about integer overflow and buffer overflow which are highly exploitable and very dangerous.

By building a fuzzer to test and analyze Android browser, this article managed to find vulnerabilities of Android browser when parsing BMP tag.

3 Fuzzing mechanisms

There are two fuzzing skills: Blind Fuzzing and Smart Fuzzing. Blind fuzzing generates mal-formed data by inserting random data in random location to form a test case. As lack of pertinence, such kind of fuzzing may generate lots of ineffective test cases when testing a sophisticated file structure and it is impossible to dig deep in the parser. Due to the flaws above, smart fuzzing based on file structure is an appropriate solution. Smart fuzzing based on file structure requires such kind of file must possess clear data structure.
What is left to do is by analyzing the data structure and information of historical vulnerabilities, to locate the vulnerable points, generate well-targeted mal-formed test cases, trigger the crashes of the parser and then analyze the reason.

3.1 Generate mal-formed data

Mal-formed Data Generator is designed to generate test cases which contain mal-formed data constructed according to the vulnerable points of BMP file structure. After analyzing all 82 BMP vulnerabilities, we found out vulnerable points occurred in the following fields: bfOffbits of BITMAPFILEHEADER, biClrUser, biWidth, biHeight and biSizeImage of BITMAPINFOHEADER. The vulnerable fields and their functions are shown in Table I:

Table I Vulnerable points of bitmap

Fields	Size	Description
bfOffBits	4Bytes	File offset from the beginning of the file to the bitmap data
biWidth	4Bytes	Bitmap width in pixels
biHeight	4Bytes	Bitmap height in pixels
biClrUsed	4Bytes	Number of actually used colors
biSizeImage	4Bytes	Size of the image data, in bytes. If there is no compression, it is valid to set this member to zero.

When constructing mal-formed data, we usually use data type as follows:

1) Arithmetic Data: Including boundary values in HEX, ASCII, Unicode, Raw. Decimal values as -2,-1,0, 2^8-2 = 254, 2^8-1 = 255,2^8 =256, 2^12-2 = 4094, 2^12-1 =4095, 2^12=4096,2^32-2=… ;Hexadecimal values as −2 = 0xFFFFFE, −1 = 0xFFFFFF…

2) The pointers: Including Null pointers, valid/invalid memory pointers.

3) String: Including extra-long strings, string without null(0x00) terminated, format string %s, %x, %25s, %n, %d …

4) Special characters: Including #, @, ', <, >, /, \, ../ …

In this article, all modules were written in Python scripts. Mal-formed Data Generator consists of three python files: "BmpFuzzer", "Structure" and "Generator". "Generator.py" is designed to generated mal-formed data as described above. "Structure.py" is designed to specify the file structure of Bitmap. "BmpFuzzer.py" is in charge of combining "Generator.py" and "Structure.py", filling the vulnerable points with mal-formed data and producing a set of HTML test cases containing mal-formed data. The reason that the first module is divided into three parts is consideration of extensibility and generality of the fuzzer, by which the type of mal-formed data and vulnerable points can be altered according to different occasions.

3.2 Input mal-formed data automatically

The input part is designed to open HTML files containing mal-formed data using Android browser. In this part, we have to use the Android emulator provided by Android SDK to establish the testing framework. In the emulator, open http://10.0.2.2 to access local web resources. For example, you can use the Tomcat server page by opening http://10.0.2.2:8080/1.html. As there are many test cases containing mal-formed data, this fuzzer adopted a simple way to perform automated testing by using JavaScript to control the jump of web page:

```
<script>
        window.open("URL.html")
        setTimeout(parent.close(),1000)
</script>
```

The "URL.html" refers to the next test case to be opened which has already been assigned when generating the test cases. This is the simplest way to carry out automation, but it will also come out with some incorrect crashes, which requires confirmation of each crash. This part is to be enhanced in future.

3.3 Monitor the exceptions

Android SDK provides a debugging tool LogCat, which can show the debugging information of application in real time and keep the information in logs. We can use this tool to monitor the crash of Android browser. Besides LogCat, this article use "LogReader.py" to read the log information of Web Server. When exceptions or crashes happen, Android browser would throw exceptions and show error warning on the screen and after a while the process will be terminated. In the meantime, LogCat will show the address of error and debugging information. Using "LogReader.py" will get the information of exception and communication between web server and browser. With all the information, we can use GDB provided by Android SDK and other debugging tools to analyze and keep deeper track of the exception or crash, confirm the reason of error and do the exploitation.

4 Implementation and experiments

4.1 System implementation

To verify the theories proposed above, we implemented an Android Browser Fuzzer in Python. The architecture of the Fuzzer is shown in Figure 2. The Fuzzer is divided into three main modules. Mal-formed Data Generator which contains three files, is to generate test cases that would be passed to the second module Mal-formed Test cases Reader which invoke the Android browser to open test cases. The third module is Exception Monitor, designed to record the exceptions and crashes information for further research.

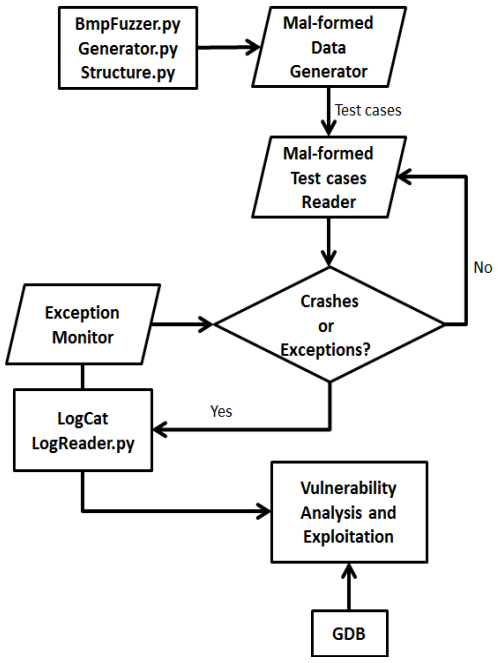

Figure 2 Android browser fuzzer

4.2 Experiments

Firstly generate 218 mal-formed test cases using "BmpFuzzer.py", then open the first test case "1.html" of web server using Android browser which leads to the automated testing. By analyzing the information and logs of exception monitoring and debugging, vulnerability has been found when browser of Android SDK m3-rc20 parses bmp files.

The information by "Exception Monitor" is as follows:
Part of Debugging information of LogCat is shown in Figure 3:

```
I/DEBUG   (  448):     bec38468  001fccf4  [heap]
I/DEBUG   (  448):     bec3846c  001ff120  [heap]
I/DEBUG   (  448):     bec38470  00080000  [heap]
I/DEBUG   (  448):     bec38474  ac0361ec  /system/lib/libsgl.so
I/DEBUG   (  448):     bec38478  0000002e
I/DEBUG   (  448):     bec3847c  ac04e7d4  /system/lib/libsgl.so
I/DEBUG   (  448):     bec38480  00000000
I/DEBUG   (  448):     bec38484  01320000
I/DEBUG   (  448):     bec38488  00000000
I/DEBUG   (  448):     bec3848c  00010000  [heap]
I/DEBUG   (  448):     bec38490  00e20000
I/DEBUG   (  448):     bec38494  bec38710  [stack]
I/DEBUG   (  448):     bec38498  00000000
I/DEBUG   (  448):     bec3849c  40000000
I/DEBUG   (  448):     bec384a0  01320000
I/DEBUG   (  448):     bec384a4  00e20000
I/DEBUG   (  448):     bec384a8  01400000
I/DEBUG   (  448):     bec384ac  00f00000
I/DEBUG   (  448):     bec384b0  01320000
I/DEBUG   (  448):     bec384b4  00e20000
I/DEBUG   (  448):     bec384b8  01400000
I/DEBUG   (  448):     bec384bc  00f00000
I/DEBUG   (  448):     bec384c0  00000132
I/DEBUG   (  448):     bec384c4  000000e2
I/DEBUG   (  448):     bec384c8  00000140
I/DEBUG   (  448):     bec384cc  000000f0
I/DEBUG   (  448):     bec384d0  00000000
I/DEBUG   (  448):     bec384d4  bec38550  [stack]
I/DEBUG   (  448):     bec384d8  00000000
I/DEBUG   (  448):     bec384dc  00191ec0  [heap]
I/DEBUG   (  448):     bec384e0  00000000
I/DEBUG   (  448):     bec384e4  bec386dc  [stack]
I/DEBUG   (  448):     bec384e8  00e20000
I/DEBUG   (  448):     bec384ec  01320000
I/DEBUG   (  448):     bec384f0  bec385f8  [stack]
I/DEBUG   (  448):     bec384f4  ac04a948  /system/lib/libsgl.so
I/DEBUG   (  448):     bec384f8  bec386dc  [stack]
I/DEBUG   (  448):     bec384fc  00191ec0  [heap]
I/ActivityManager(  461): APP DEATH: com.google.android.browser
```

Figure 3 Debugging information of LogCat

Log information of web server by "LogReader.py" is shown in Figure 4:

```
127.0.0.1 - - "GET /fuzzerbmp/1.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/2.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/3.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/4.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/5.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/6.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/7.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/8.bmp HTTP/1.1" 200 311
127.0.0.1 - - "GET /fuzzerbmp/9.bmp HTTP/1.1" 200 311
```

Figure 4 Log of web server

As shown above when the browser opened "9.bmp", an exception occurred and application was terminated. After opening "9.bmp" using 010Editor, the data of different fields are shown in Figure 5:

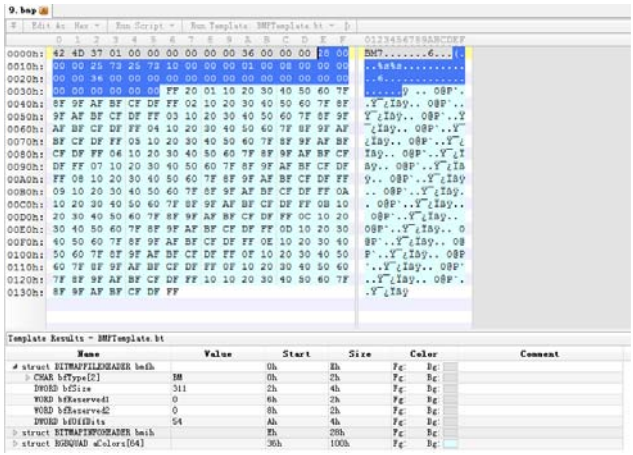

Figure 5 Target BMP file data

"Exception Monitor" kept the value of registers R1-R9, PC,SP and states of heap and stack, as well as the fault address, which would be of great help for further analysis and exploitation.

5 Conclusions

As an import method of vulnerability analyzing, fuzzing is very effective. With rapid development of fuzzing technologies, logic-oriented, data type oriented and sample oriented smart fuzzing is becoming the focus of researchers. Android operation system has adopted a lot of open source codes which may contain many vulnerabilities which would affect the Android operating system itself. This article described a fuzzing method based on smart fuzzing theories and constructed a fuzzer with several modules, which verified a historical vulnerability (CVE-2008-0986) and discovered an unknown vulnerability. The result of the experiment proved the correctness of the fuzzing method and the effectiveness of the fuzzer, and also provided anther evidence that there is security flaw in Android system and it is worth doing further research.

Acknowledgement

Natural Science Foundation of China (No.90818021, No. 61272493, No. 61100047)

References

[1] comScore. U.S. Mobile Subscriber Market Share. 2012.
[2] Eguan.cn. Mobile Terminal Market Share of Q1 2012.
[3] Davi, L., Dmitrienko, A., Sadeghi, A.R. and Winandy, M.Privilege escalation attacks on Android [J]. Information Security. 340-360.
[4] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y. and Dolev, S. Google Android: A state-of-the-art review of security mechanisms [J]. Arxiv preprintarXiv: 0912.5101.
[5] Chaudhuri, A. Language-based security on Android[C]. Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, 2009, 1-7.
[6] Qing Wang, Dong-hui Zhang, Hao Zhou et al. 0day Security: Skills of Software Vulnerability Analysis. Publishing House of Electronics Industry. 414-420.
[7] Michael Sutton. The Art of File Format Fuzzing [DB/OL].http://www.blackhat.com/presentations/bh-us-05-sutton.pdf
[8] Feng-sheng Yang. Android Knowledge [M].China Machine Press. 258-260.